AD-A043 450 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
NONDETERMINISM AND THE CORRECTNESS OF PARALLEL PROGRAMS.(U)
MAY 77 L FLON, N SUZUKI

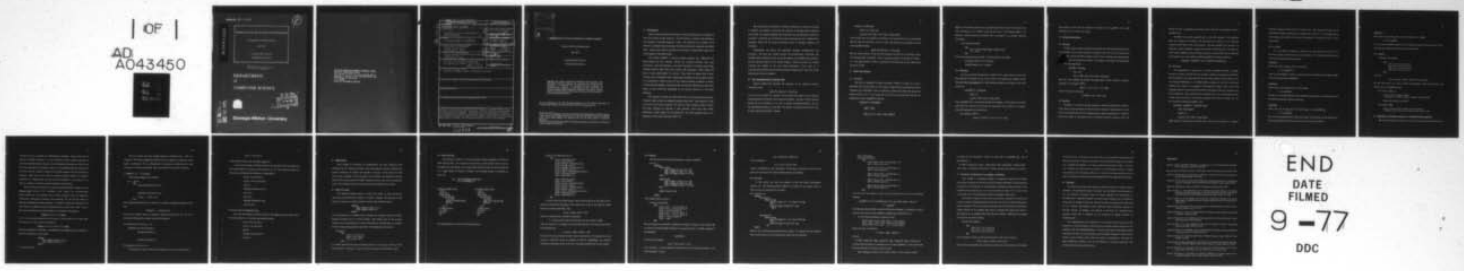
UNCLASSIFIED

F44620-73-C-0074

AFOSR-TR-77-1137

NL

| OF |
AD
A043450



END
DATE
FILMED
9-77
DDC

Nondeterminism and the Correctness of Parallel Programs

Lawrence Flon[1] and Norihisa Suzuki

May 1977

Carnegie-Mellon University

Pittsburgh, Pennsylvania

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR- 77-1137	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): NONDETERMINISM AND THE CORRECTNESS OF PARALLEL PROGRAMS	5. TYPE OF REPORT & PERIOD COVERED: Interim rept.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Lawrence Flon and Norihisa Suzuki	8. CONTRACT OR GRANT NUMBER(s): F44620-73-C-0074, NSF-DCR-74-24573
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Carnegie-Mellon University, Computer Science Dept., Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F, 2304/A2, A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd, Arlington, VA 22209	12. REPORT DATE: May 77
	13. NUMBER OF PAGES: 21
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Air Force of Scientific Research (NM), Bolling AFB, DC 20332	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper presents the weakest preconditions which describe weak correctness, blocking, deadlock, and starvation for nondeterministic programs. A procedure for converting parallel programs to nondeterministic programs is described, and the correctness of various example parallel programs is treated in this manner. Among these are a busy-wait mutual exclusion scheme, and the problem of the Five Dining Philosophers.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403 081

Nondeterminism and the Correctness of Parallel Programs

Lawrence Flon[1] and Norihisa Suzuki

May 1977

Carnegie-Mellon University

Pittsburgh, Pennsylvania

Abstract: We present weakest pre-conditions which describe weak correctness, blocking, deadlock, and starvation for nondeterministic programs. A procedure for converting parallel programs to nondeterministic programs is described, and the correctness of various example parallel programs is treated in this manner. Among these are a busy-wait mutual exclusion scheme, and the problem of the Five Dining Philosophers.

To be presented at the IFIP Working Conference on the Formal Description of Programming Concepts, St. Andrews, New Brunswick, Canada, Aug. 1-5.

This work was supported in part by the Defense Advanced Research Projects Agency under contract no. F44620-73-C-0074, and in part by the National Science Foundation under grant DCR74-24573, and monitored by the Air Force Office of Scientific Research.

[1]Author's address 8/77: Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712.

1. Introduction

Some of the most difficult to find bugs in systems programs arise in situations of concurrent access to data structures. Thus the need for a precise understanding of the semantics of parallel programs is clear. The attainment of a suitable formal definition of parallel program semantics will allow construction of automatic verification tools. These should help to eliminate the frustrating, "irreproducible" bugs which usually plague an operating system.

The earliest attempts at verifying parallel programs, e.g. [Habermann 72, Brinch Hansen 72] were basically informal and concerned primarily with weak correctness. [Flon 76] describes a semi-formal approach to verifying concurrently accessed abstract data types which contain path expressions. Hoare [Hoare 74] gave a formal axiomatization for monitors. These latter two papers take a data-oriented view of parallelism which, though quite reasonable for the problems treated, is not particularly suited for proof of such strong correctness properties as safety from blocking and deadlock. By data-oriented we mean with regard only to concurrent access to data structures, independent of the control structures of the actual processes.

The approach of Owicki and Gries [Owicki 76] is a process-oriented extension of Hoare's axiom system for sequential programs [Hoare 69] - close attention is paid to the form of the actual processes. Our goals are closely related to those of Owicki and Gries, although our approach is quite different. There have seen similar approaches, notably [Keller 76, Lamsveerde 76]. The work presented here is an outgrowth of some ideas discussed in [Flon 77].

We shall begin by using Dijkstra's predicate transformers, primarily the concept of weakest pre-condition, to describe the semantics of *nondeterministic* programs which differ from Dijkstra's [Dijkstra 76] in that they are not necessarily required to terminate. In particular we will discuss the weak correctness of such nondeterministic programs, along with the strong correctness issues of blocking, deadlock, and starvation.

Subsequently we discuss the relationship between nondeterminism and parallelism. We show how parallel programs can be effectively transformed into nondeterministic programs, so that the results obtained for nondeterministic programs can be indirectly used to verify parallel programs. Several examples are treated, including the problem of the Five Dining Philosophers, which relies on a synchronization primitive, and an old Mutual Exclusion scheme which relies only on the indivisibility of access to memory.

2. The Nondeterministic Command Rep

Dijkstra [Dijkstra 76] describes the semantics of the repetitive guarded command, written

$$\underline{\text{do}}\ B_1 \rightarrow S_1 \parallel B_2 \rightarrow S_2 \parallel \ldots B_n \rightarrow S_n\ \underline{\text{od}}$$

in terms of the weakest (i.e. necessary and sufficient) pre-condition to the command which guarantees termination with a given post-condition. The intent of DO is that the guards B_j are evaluated, a true one is chosen (nondeterministically), and its corresponding statement S_j is executed. The process is continued until no B_j is true, at which point DO terminates. Formally,

$wp(DO, R) = (\exists k \geq 0) H_k(R)$

$H_0(R) = R \wedge \neg(\exists j \in 1..n) B_j$

$H_{k+1}(R) = (\exists j \in 1..n) B_j \wedge (\forall j \in 1..n)(B_j \Rightarrow wp(S_j, H_k(R)))$

where $H_k(R)$ gives the weakest pre-condition which assures that R will be established and the loop will terminate in at most k steps. We define for our purposes a similar command, REP, denoted

$$\underline{rep}\ B_1 \rightarrow S_1 \parallel B_2 \rightarrow S_2 \parallel \ldots B_n \rightarrow S_n\ \underline{per}$$

REP has similar behavior to DO, with the exception that REP will not terminate unless the command "exit" is executed. Thus, if no guard evaluates to true, REP will "hang" - i.e. it will <u>not</u> terminate. If there is in general no final state, how are we to define the semantics of REP?

3. Weak Correctness

3.1 Invariance

One way to represent the weak correctness of REP is in terms of a set of assertions $\{P_j \mid 1 \leq j \leq n\}$, where n is the number of guards and P_j is guaranteed to hold whenever S_j is executable. That is, a selection is about to be made from among the guards and B_j is true. If we denote the fact that P must always hold whenever B_j evaluates to true as $wlp_j(REP, P)$, we have

$wlp_j(REP, P) = (\forall k \geq 0) W_k^j(P)$

$W_0^j(P) = B_j \Rightarrow P$

$W_{k+1}^j(P) = (\forall i \in 1..n)(B_i \Rightarrow wlp(S_i, W_k^j(P)))$

$W^j_k(P)$ is the weakest condition which guarantees $B_j \Rightarrow P$ after exactly k statements have been executed, if it is possible to execute that many. The predicate *wlp*(S, R) is Dijkstra's weakest liberal pre-condition, and is equivalent to *wp* without requiring termination.

As an example, consider

```
rep
      I=0 → x←x/2; if odd(x) then I←1 else skip fi ||
      I=1 → x←x-1
per
```

Let P_1 be even(x) and P_2 be odd(x). Then by the above recurrence,

$$wlp_1(REP, even(x)) = (I=0 \Rightarrow even(x))$$

$$wlp_2(REP, odd(x)) = (I=1 \Rightarrow odd(x))$$

3.2 Potentiality

Let us also consider the question of whether or not a given guard can ever (has the potential to) evaluate to true. That is, what is the weakest pre-condition to REP that guarantees the existence of a finite execution sequence which leads to the truth of B_j? Let

$$wpot(REP, P) = (\exists k \geq 0) V_k(P)$$

$$V_0(P) = P$$

$$V_{k+1}(P) = (\exists j \in 1..n)(B_j \wedge wp(S_j, V_k(P)))$$

Then *wpot*(REP, B_j) is the desired weakest pre-condition. In the above recurrence, $V_k(P)$ gives the weakest pre-condition that guarantees the existence of a length-k execution sequence which establishes P.

For example, if REP is

```
rep A → x←0 || B → x←1 || x=1 → x←x+1 per
```

then guard 3 (x=1) has the potential to become true iff *wpot*(REP, x=1), which evaluates to (x=1 ∨ B), is true at entry.

4. Strong Correctness

4.1 Blocking

Neither wlp_j nor *wpot* is sufficient to guarantee that REP will do anything useful. For example, it may be that eventually no guards will be true and the command will "hang". We shall say that REP is *blocked* if such a state is reached, and is *blocking-free* if no such state can possibly be reached. The weakest pre-condition that guarantees that REP is blocking-free is

$$wbp(REP) = (\forall k \geq 0) G_k$$

$$G_0 = true$$

$$G_{k+1} = (\exists j \in 1..n) B_j \wedge (\forall j \in 1..n)(B_j \Rightarrow wp(S_j, G_k))$$

Here, G_k is the weakest pre-condition that guarantees at least a length-k execution sequence. For example, if REP is

$$\underline{rep}\ x>y \rightarrow x \leftarrow x-y \parallel y>x \rightarrow y \leftarrow y-x\ \underline{per}$$

then by the above recurrence

$$wbp(REP) = x \neq y \wedge (x \leq 0 \vee y \leq 0)$$

4.2 Deadlock

Deadlock in a system of parallel processes is defined in [Holt 72] as a state in which "one or more processes are blocked forever because of requirements that can never be satisfied." We will call a *nondeterministic* program *deadlocked* if it reaches a state from which, for any guard, there is no possible execution sequence which will

lead to its truth. A *deadlock-free* program is one in which it is not possible to reach a deadlock state.

Consider the predicate *wpot*(REP, B_j) for some REP command. That predicate gives the weakest pre-condition that guarantees the existence of an execution sequence which leads to the truth of guard j. Suppose *wpot*(REP, B_j) is always true whenever a guard selection is made, and that REP is blocking-free. Then REP can never reach a deadlock state with respect to guard j, since B_j always has the potential to evaluate to true. This condition, denoted wdp_j(REP) is precisely defined by

$$wdp_j(REP) = wbp(REP) \wedge (\forall k \in 1..n) wlp_k(REP, wpot(REP, B_j))$$

4.3 Starvation

The phenomenon of *starvation* in a system of parallel processes is another strong correctness issue that we must consider in addition to blocking and deadlock. Dijkstra [Dijkstra 71] briefly discusses this issue with respect to the problem of the Five Dining Philosophers. In a nondeterministic program, we say that a particular statement may *starve* if it is possible for the program to reach a state in which the statement's guard is false, and state transitions which leave it false may continue to be taken indefinitely. Thus a given guard j is starvation-free if it is not possible to reach a state from which there is an execution sequence which forever maintains $\neg B_j$. Let this condition be denoted wsp_j(REP). Then

$$wsp_j(REP) = wbp(REP) \wedge \neg wpot(REP, U(\neg B_j))$$

$$U(R) = (\forall k \geq 0)\ U_k(R)$$

$$U_0(R) = R$$

$$U_{k+1}(R) = (\exists i \in 1..n)(B_i \wedge wp(S_i, U_k(R)))$$

$U_k(R)$ gives the weakest pre-condition which assures the existence of a length-k

execution sequence during which R is always true. U(R) requires that there be an unbounded execution sequence which maintains the truth of R. Thus $wsp_j(REP)$ gives the weakest pre-condition which denies the possibility of reaching a state in which $U(\neg B_j)$ holds.

4.4 Invariants

It is not always necessary to compute all of the stated recurrences in the previous sections in order to verify a given program. The following theorems follow from the previously defined recurrences:

Theorem 1

$$[(\forall k \in 1..n)(\mathcal{J} \wedge B_k \Rightarrow wlp(S_k, \mathcal{J}))] \wedge (\mathcal{J} \Rightarrow (B_j \Rightarrow P))$$
$$\vdash \mathcal{J} \Rightarrow wlp_j(REP, P)$$

That is, any predicate $\mathcal{J}$ which is invariant across each statement and implies $(B_j \Rightarrow P)$ will suffice to guarantee $wlp_j(REP, P)$.

Theorem 2

$$[(\forall k \in 1..n)(\mathcal{J} \wedge B_k \Rightarrow wp(S_k, \mathcal{J})] \wedge (\mathcal{J} \Rightarrow (\exists k \in 1..n) B_k)$$
$$\vdash \mathcal{J} \Rightarrow wbp(REP)$$

Similarly, an invariant predicate which implies the existence of a true guard must be sufficient to guarantee absence of blocking.

Theorem 3

$$[(\forall k \in 1..n)(\mathcal{J} \wedge B_k \Rightarrow wp(S_k, \mathcal{J})] \wedge (\mathcal{J} \Rightarrow (\exists k \in 1..n) B_k) \wedge (\mathcal{J} \Rightarrow wpot(REP, B_j))$$
$$\vdash \mathcal{J} \Rightarrow wdp_j(REP)$$

Any invariant predicate which implies both safety from blocking and the potential for B_j to be established must guarantee guard j to be safe from deadlock.

Theorem 4

$$[(\forall k \in 1..n)(\mathcal{J} \wedge B_k \Rightarrow wp(S_k, \mathcal{J})] \wedge (\mathcal{J} \Rightarrow (\exists k \in 1..n) B_k) \wedge (\mathcal{J} \Rightarrow \neg U(\neg B_j))$$

$$\vdash \mathcal{J} \Rightarrow wsp_j(REP)$$

If an invariant predicate implies that no unbounded execution sequence exists which keeps B_j false, then guard j must be free from starvation.

4.5 Example

Consider the command

```
rep
      x>0 ∧ y<n → x←x-1; y←y+1 ||
      y>0 ∧ z<n → y←y-1; z←z+1 ||
      z>0 ∧ w<n → z←z-1; w←w+1 ||
      w>0 ∧ x<n → w←w-1; x←x+1
per
```

Let $\mathcal{J}$ be

$$0 \le x \le n \wedge 0 \le y \le n \wedge 0 \le z \le n \wedge 0 \le w \le n \wedge 0 < x+y+z+w < 4n$$

We will show that if $\mathcal{J}$ is true at entry, the command is safe from blocking. By Theorem 2, $\mathcal{J} \Rightarrow wbp(REP)$ because

1) $\mathcal{J} \wedge B_1 \Rightarrow wp(S_1, \mathcal{J})$

$$0 < x \le n \wedge 0 \le y < n \wedge 0 \le z \le n \wedge 0 \le w \le n$$
$$\Rightarrow 1 \le x \le n+1 \wedge -1 \le y \le n-1 \wedge 0 \le z \le n \wedge 0 \le w \le n \wedge 0 < x+y+z+w < 4n$$

(the others follow by symmetry)

2) $\mathcal{J} \Rightarrow (\exists k \in 1..n) B_k$

$$0 \le x \le n \wedge 0 \le y \le n \wedge 0 \le z \le n \wedge 0 \le w \le n \wedge 0 < x+y+z+w < 4n$$
$$\Rightarrow (X>0 \wedge y<n) \vee (y>0 \wedge z<n) \vee (z>0 \wedge w<n) \vee (w>0 \wedge x<n)$$

5. Reduction of parallel programs to nondeterministic programs

We have seen that such problems as weak correctness, blocking, deadlock, and

starvation can be formalized for nondeterministic programs. These results can be applied to parallel programs if we can effectively convert parallel programs to equivalent nondeterministic programs. Here the meaning of equivalence is that for any execution sequence of one program, there is a corresponding execution sequence of the other such that values of variables in the parallel program have the same history sequence. (Note that we will have to introduce program counters to transform parallelism to nondeterminism, so they are only equivalent in this sense.) For a rigorous treatment of models of parallel computation see [Karp 69].

We will outline a procedure for transforming parallel programs made up of the cobegin-coend construct with conditional critical regions for synchronization [Brinch Hansen 73]. The sequential parts of these programs consist of assignment, conditionals, while-loops, compounds, and sequencing. We will use the notion of *effective indivisibility* of program segments. A segment is effectively indivisible if the final values of variables are always determined only by their initial values. That is, they are not affected by the other processes. For example, in the program

cobegin x←x+1 // x←x+1 coend

x←x+1 is not indivisible because the final value of x may be either 1 or 2 more than the initial value. If we convert the program to

cobegin c←x; x←c+1 // d←x; x←d+1 coend

then each assignment is indivisible. The conditional critical region itself is by definition effectively indivisible, so the program

```
cobegin
      with x when true do x←x+1 //
      with x when true do x←x+1
coend
```

is in indivisible form.

We first convert the entire parallel program to indivisible form - that is, a program in which every assignment statement which is outside of a conditional critical region is indivisible. This is accomplished by introducing variables local to each process as in the previous example. Next, each statement is converted as follows:

1) cobegin P_1 // . . .// P_n coend

The program skeleton is converted to

$p_1 \leftarrow \ldots p_n \leftarrow 0$;
rep
<guarded commands for P_1> ||
.
.
.
<guarded commands for P_n> ||
$c_1 = m_1 \wedge \ldots c_n = m_n \rightarrow$ exit
per

Here, <guarded commands for P_i> consists of a number of guarded commands of the form

<condition> → <statement list>

and m_i is the largest value of c_i assigned in <guarded commands for P_i>. The "exit" command is introduced to provide a means for termination.

2) Sequencing of the form $S_1; \ldots S_n$

Statement lists are converted to

<guarded command> ||
.
.
.
<guarded command> ||

3) Assignment of the form $x \leftarrow e$

If the program counter for the previous statement is n, this is transformed to

$p_j{=}n{+}1 \rightarrow x \leftarrow e;\ p_j \leftarrow n{+}2$

4) Conditionals of the form if B then S_1 else S_2 fi

Let k_1 be the number of different values of the program counter resulting from the transformation of S_1, and k_2 be the same for S_2. If n is the program counter for the previous statement, the conditional is transformed to

$p_j{=}n{+}1 \wedge B \rightarrow p_j \leftarrow n{+}2 \ \|$

$p_j{=}n{+}1 \wedge \neg B \rightarrow p_j \leftarrow n{+}k_1{+}2 \ \|$

$p_j{=}n{+}2 \ldots$

<guarded commands for S_1>

$p_j{=}n{+}1{+}k_1 \ldots$

$p_j{=}n{+}2{+}k_1 \ldots$

<guarded commands for S_2>

$p_j{=}n{+}1{+}k_1{+}k_2 \ldots$

5) Loops of the form while B do S od

Let k be the number of different values of the program counter which result from converting S. If n is the previous program counter,

$p_j{=}n{+}1 \wedge B \rightarrow p_j \leftarrow n{+}2 \ \|$

$p_j{=}n{+}1 \wedge \neg B \rightarrow p_j \leftarrow n{+}2{+}k$

$p_j{=}n{+}2 \ldots$

<guarded commands for S>

$p_j{=}n{+}1{+}k \ldots$

6. Application

Even though by conversion to nondeterminism we have obtained formal definitions for the various correctness issues associated with parallel programs, the question remains as to whether this approach is practical. In this section we shall treat various examples to show the power of our method. The approach of Owicki [Owicki 75] is both practical and highly dissimilar to ours, so by way of comparison some of the examples we discuss have been previously handled by her method.

6.1 Weak correctness

The following example appears in [Owicki 75], where its weak correctness cannot be proved without the addition of auxiliary variables. The discovery of the right set of auxiliary variables (in general) requires much intellectual effort.

```
cobegin
    with x when true do x←x+1 //
    with x when true do x←x+1
coend
```

The transformation to a nondeterministic program also introduces auxiliary variables (program counters), but in a uniform manner. Even though some of the program counters may be superfluous, we remove the burden of inventing auxiliary variables and the corresponding operations upon them. The nondeterministic version is

```
p1←p2←0;
    rep
        p1=0 → x←x+1; p1←1 ||
        p2=0 → x←x+1; p2←1 ||
        p1=1 ∧ p2=1 → exit
    per
```

It is easily seen that the weak correctness invariant is $x=x_0+p_1+p_2$, where x_0 is the initial value of x. Since $p_1=1 \wedge p_2=1$ at the exit, the program will establish $x=x_0+2$.

6.2 Mutual exclusion

We consider a solution to a mutual exclusion problem discussed by Dijkstra in 1965 [Dijkstra 65]. Two processes A and B have critical sections which must be excluded from one another. No synchronization primitive other than the indivisibility of a single access to memory is allowed. The following solution is discussed in [Flon 77]:

```
var   inA, inB: boolean initially false,
      prty: (A,B) initially A;
```

```
processA: while true do
   <think>
   inA←true;
   while inB do
       if prty=B then
           inA←false;
           while prty=B do skip od;
           inA←true
           fi
       od;
   <critical section>
   inA←false;
   prty←B
   od
```

```
processB: while true do
   <think>
   inB←true;
   while inA do
       if prty=A then
           inB←false;
           while prty=A do skip od;
           inB←true
           fi
       od;
   <critical section>
   inB←false;
   prty←A
   od
```

The nondeterministic version of this parallel system is

```
p1←p2←1; inA←inB←false; prty←A;
    rep
        p1=1 → inA←true; p1←2 ||
        p1=2 ∧ inB → p1←3 ||
        p1=3 ∧ prty=B → inA←false; p1←4 ||
        p1=4 ∧ prty=B → skip ||
        p1=4 ∧ prty≠B → inA←true; p1←2 ||
        p1=3 ∧ prty≠B → p1←2 ||
        p1=2 ∧ ¬inB → <critical section>; p1←5 ||
        p1=5 → inA←false; p1←6 ||
        p1=6 → prty←B; p1←1 ||

        p2=1 → inB←true; p2←2 ||
        p2=2 ∧ inA → p2←3 ||
        p2=3 ∧ prty=A → inB←false; p2←4 ||
        p2=4 ∧ prty=A → skip ||
        p2=4 ∧ prty≠A → inB←true; p2←2 ||
        p2=3 ∧ prty≠A → p2←2 ||
        p2=2 ∧ ¬inA → <critical section>; p2←5 ||
        p2=5 → inB←false; p2←6 ||
        p2=6 → prty←A; p2←1 ||
    per
```

To show that the critical sections cannot both be active at the same time, it suffices to prove that the guards which reflect entry to the critical sections cannot both be true at the same time. Thus,

$$(p1=2 \wedge \neg inB) \wedge (p2=2 \wedge \neg inA)$$

must be invariantly false. Consider the predicate

$$L = (p1=3 \Rightarrow inA) \wedge (p2=3 \Rightarrow inB) \wedge [(p1=2 \Rightarrow inA) \vee (p2=2 \Rightarrow inB)]$$

(The last conjunct is the negation of the previous formula.) L is invariant across all of the commands, and

$$L \Rightarrow \neg(p1=2 \wedge \neg inB) \vee \neg(p2=2 \wedge \neg inA)$$

so clearly the critical sections cannot be active simultaneously. This program is proved correct in [Flon 77], using an extension of Owicki's methodology, but auxiliary variables are needed in a non-trivial way. The proof presented here is much simpler.

6.3 Deadlock

We will show that the following program is subject to deadlock:

```
r1←r2←1;
      cobegin
            while true do
                  with r2 when r2=1 do r2←r2-1 od;
                  with r1 when r1=1 do r1←r1-1 od;
                  with r1,r2 when true do r1←r2←1 od
            od //

            while true do
                  with r1 when r1=1 do r1←r1-1 od;
                  with r2 when r2=1 do r2←r2-1 od;
                  with r1,r2 when true do r1←r2←1 od
            od //

            while true do skip od

      coend
```

The nondeterministic version is

```
r1←r2←1; p1←p2←p3←0;
      rep
            p1=0 ∧ r1=1  →  r1←r1-1; p1←1  ||
            p1=1 ∧ r2=1  →  r2←r2-1; p1←2  ||
            p1=2  →  r1←r2←1; p1←0  ||
            p2=0 ∧ r2=1  →  r2←r2-1; p2←1  ||
            p2=1 ∧ r1=1  →  r1←r1-1; p2←2  ||
            p2=2  →  r1←r2←1; p2←0  ||
            p3=0  →  skip
      per
```

We can establish the possibility of deadlock by finding an invariant $\mathcal{J}$ which implies $\neg B_j$ for some j, and then showing that there is a way to arrive at $\mathcal{J}$. The latter property is expressed by

$$\text{wpot}(\text{REP}, \mathcal{J})$$

For the above program

$$p1=1 \wedge r2=0 \wedge p2=1 \wedge r1=0$$

is an invariant. It is also possible to achieve this by first executing command 1 and then command 4. That is,

$$B_1 \wedge \text{wlp}(S_1, B_4 \wedge \text{wlp}(S_4, \mathcal{J}))$$

which evaluates to

$$r1{=}1 \wedge r2{=}1 \wedge p1{=}0 \wedge p2{=}0$$

which is established by the initialization. Furthermore, $\mathcal{J}$ prevents all but the last guard from running, so the original parallel program may deadlock.

6.4 Starvation

In this section we treat the problem of the Five Dining Philosophers [Dijkstra 71]. The following parallel program is a solution to the problem which is known to have the possibility of starvation:

```
f0←f1←f2←f3←f4←1;
    cobegin
        philosopher 1:
            while true do
                with f0,f1 when f0=1 ∧ f1=1 do f0←f1←0 od;
                "eat";
                with f0,f1 when true do f0←f1←1 od
                  od //
            .
            .
            .
        philosopher 5:
            while true do
                with f4,f0 when f4=1 ∧ f0=1 do f4←f0←0 od;
                "eat";
                with f4,f0 when true do f4←f0←1 od
              od
    coend
```

Below is the corresponding nondeterministic program. It is evident that the program does not terminate, so we have deleted the superfluous exit statement.

```
f0←f1←f2←f3←f4←1;
p1←p2←p3←p4←p5←0;
      rep
            p1=0 ∧ f0=1 ∧ f1=1 → f0←f1←0; p1←1  ||
            p1=1 → f0←f1←1; p1←0  ||

            p2=0 ∧ f1=1 ∧ f2=1 → f1←f2←0; p2←1  ||
            p2=1 → f1←f2←1; p2←0  ||

            p3=0 ∧ f2=1 ∧ f3=1 → f2←f3←0; p3←1  ||
            p3=1 → f2←f3←1; p3←0  ||

            p4=0 ∧ f3=1 ∧ f4=1 → f3←f4←0; p4←1  ||
            p4=1 → f3←f4←1; p4←0  ||

            p5=0 ∧ f4=1 ∧ f0=1 → f4←f0←0; p5←1  ||
            p5=1 → f4←f0←1; p5←0  ||
      per
```

To prove that the program is not free from starvation, we will use the following theorem:

$$wpot(REP, P) \wedge (P \Rightarrow wpot(REP, B_j)) \wedge [P \Rightarrow \neg B_j \wedge (\exists k \in 1..n)(B_k \wedge wp(S_k, P)] \vdash \neg wsp_j(REP)$$

The theorem states that REP is subject to starvation whenever it is possible to reach a state P, from which it is both possible to establish B_j and possible not to.

For the Dining Philosophers program, let P be

```
(f4=0 ∧ f0=0 ∧ f1=0 ∧ f2=0 ∧ p5=1 ∧ p1=0 ∧ p2=1) ∨
(f4=1 ∧ f0=1 ∧ f1=0 ∧ f2=0 ∧ p5=0 ∧ p1=0 ∧ p2=1) ∨
(f4=0 ∧ f0=0 ∧ f1=1 ∧ f2=1 ∧ p5=1 ∧ p1=0 ∧ p2=0)
```

Clearly $P \Rightarrow \neg B_1$. Furthermore,

$$P \Rightarrow (\exists k \in 1..n)(B_k \wedge wp(S_k, P))$$

because

$$P \Rightarrow (B_3 \wedge wp(S_3, P)) \vee (B_4 \wedge wp(S_4, P)) \vee (B_9 \wedge wp(S_9, P)) \vee (B_{10} \wedge wp(S_{10}, P))$$

To prove that the state P is reachable, we must show wpot(REP, P). This clearly holds because statement 3 may be executed first, and

$$B_3 \wedge wp(S_3, P) = p2=0 \wedge p1=0 \wedge p5=0 \wedge ((f4=1 \wedge f0=1) \vee (f4=0 \wedge f0=0))$$

is implied by the initialization. Finally, we show that $P \Rightarrow wpot(REP, B_1)$. This is guaranteed by

$$P \Rightarrow (B_4 \wedge wp(S_4, B_1)) \vee (B_{10} \wedge wp(S_{10}, B_1)) \vee (B_4 \wedge wp(S_4, B_{10} \wedge wp(S_{10}, B_1)))$$

Thus, there is a possibility that process 1 (and by symmetry any process) may starve.

7. Practical considerations for program verification

Even though it is sometimes possible to compute the weakest pre-condition recurrences, in general we will have to use less complex techniques if parallel program verification is to be practical. If we can discover sufficiently strong invariants for the nondeterministic programs, we can use the theorems of section 4.4 to verify weak correctness, safety from blocking, and safety from deadlock rather easily.

Verification of safety from starvation can be done by showing that starting from a given invariant, all possible execution sequences must arrive at a state satisfying the guard in question. This may be done by proving safety from blocking and defining an integer function of the program state which is bounded from below and which is decreased by every statement other than the one in question. Following is an example of a proof of starvation-freeness.

Consider the program

```
rep
      x>0 ∧ y<n → x←x-1; y←y+1 ||
      y>0 ∧ z<n → y←y-1; z←z+1 ||
      z>0 ∧ x<n → z←z-1; x←x+1 ||
per
```

For this program, a loop invariant which guarantees safety from blocking is

$$0 \le x \le n \wedge 0 \le y \le n \wedge 0 \le z \le n \wedge 0 < x+y+z < 3n$$

Since all three statements are symmetric, we need only prove absence of starvation

for the first one. To do that we will show that it is not possible for the second and the third processes to execute continuously without eventually making the first guard true. Suppose the program is in a state in which the first guard is false - i.e. $x=0 \vee y=n$. Consider the state function $W = 3y+2z+x$. W is decreased by exactly 1 by both the second and third statements. When $x=n \wedge y=0$, W can no longer decrease and the last two statements cannot execute, guaranteeing that the first statement will execute.

8. Summary

For some time we have been lacking an effective formalism for parallel program semantics. The approach discussed in this paper was motivated by two observations - that the important correctness issues for parallel programs have counterparts in nondeterministic sequential programs, and that parallel programs can be effectively transformed to nondeterministic ones. We have therefore presented formal definitions for the weakest pre-conditions which guarantee weak correctness, absence of blocking, absence of deadlock, and absence of starvation for nondeterministic programs, along with a procedure for the conversion of parallel programs to nondeterminism.

As a demonstration of the usefulness of our formalism we have proved various properties of several programs, including a busy-wait mutual exclusion scheme and the problem of the Five Dining Philosophers. It remains to be seen to what degree these techniques will apply to actual operating system examples, although we have tried to present various methods which reduce the burden of computation in exchange for some intellectual creativity, such as the discovery of invariant predicates and monotonically decreasing state functions.

References

[Brinch Hansen 72] Brinch Hansen, P., A Comparison of Two Synchronizing Concepts. *Acta Informatica 1,3* (1972), 190-199.

[Brinch Hansen 73] Brinch Hansen, P., Operating Systems Principles, Prentice Hall, 1973.

[Dijkstra 65] Dijkstra, E. W., Solution of a Problem in Concurrent Programming Control. *Comm. ACM 8,9* (Sept. 1965), 569.

[Dijkstra 71] Dijkstra, E. W., Hierarchical Ordering of Sequential Processes. In *Operating Systems Techniques*, Hoare and Perrot (eds.), Academic Press, London, 1971.

[Dijkstra 76] Dijkstra, E. W., A Discipline of Programming, Prentice Hall, 1976.

[Flon 76] Flon, L. and Habermann, A. N., Towards the Construction of Verifiable Software Systems. Proceedings of the ACM Conference on Data: Abstraction, Definition and Structure, *SIGPLAN Notices 8,2* (March 1976), 141-148.

[Flon 77] Flon, L., On the Design and Verification of Operating Systems. Ph.D thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (May 1977).

[Habermann 72] Habermann, A. N., Synchronization of Communicating Processes. *Comm. ACM 15,3* (March 1972), 171-176.

[Hoare 69] Hoare, C. A. R., An Axiomatic Basis for Computer Programming. *Comm. ACM 12,10* (Oct. 1969), 576-580.

[Hoare 74] Hoare, C. A. R., Monitors: An Operating System Structuring Concept. *Comm. ACM 17,10* (Oct. 1974), 549-557.

[Holt 72] Holt, R. C., Some Deadlock Properties of Computer Systems. *Computing Surveys 4,3* (Sept. 1972), 179-196.

[Karp 69] Karp, R. M. and Miller, R. E., Parallel Program Schemata. *Journal of Computer and System Science 3* (1969), 147-195.

[Keller 76] Keller, R. M., Formal Verification of Parallel Programs. *Comm. ACM 19,7* (July 1976), 371-384.

[Lamsveerde 76] van Lamsveerde, A. and Sintzoff, M., Formal Derivation of Strongly Correct Parallel Programs. MBLE Research Report, Brussels, Belgium (1976).

[Owicki 75] Owicki, S., Axiomatic Proof Techniques for Parallel Programs. Ph.D. Thesis, Department of Computer Science, Cornell University (July 1975).

[Owicki 76] Owicki, S. and Gries, D., Verifying Properties of Parallel Programs: An Axiomatic Approach. *Comm. ACM 19,5* (May 1976), 279-285.